Práctica 1. Pruebas y calidad del software

AMPLIACIÓN DE INGENIERÍA DEL SOFTWARE

4.º GII + Matemáticas

Borja Dosuna Hernández

Jaime González Casero

05-04-2021

URJC Escuela Técnica Superior de Ingeniería Informática

1. Objetivos

La presente práctica pretende implementar ciertos controles de calidad sobre una aplicación Spring Boot proporcionada encargada de gestionar una librería online, que cuenta con una interfaz web y con una API REST para la gestión de libros.

El objetivo principal será el control de calidad por dos vías principales: por medio de **pruebas automáticas** y por medio de **análisis estático del código**. Las pruebas automáticas consistirán en **test unitarios de la lógica de la aplicación, tests E2E de la API REST** (mediante el framework RESTAssured) y **tests E2E de la interfaz web** (mediante el framework Selenium). En cuanto al análisis estático del código, se analizarán los *issues* reportados por la herramienta **SonarQube** (versión 8.7.1.42226), siempre corrigiendo los de tipo *"Vulnerability"* y decidiendo si los de tipo *"Code Smell"* deben o no ser corregidos.

En cuanto a los objetivos secundarios, se perseguirá la modularización de los tests entre sí; evitando además en los unitarios la instanciación de clases DOC para probar el SUT *BookService* de forma aislada (para ello se usarán mocks). También se describe en este documento el funcionamiento de los tests, la justificación de las decisiones tomadas en el análisis de los *issues* y el procedimiento llevado a cabo en la resolución de estos.

2. Pruebas automáticas

2.1. Tests unitarios

Dentro del proyecto Maven, la clase de tests unitarios `BookServiceTest.java` pertenece al paquete `es.urjc.code.daw.library.test.unit`. Mediante *test fixtures* instanciamos la clase *BookService* y mockeamos las dependencias que necesita antes de cada test. Cada test unitario requerido en la práctica conforma un método:

- `givenBookService_whenABookIsAdded_thenBookIsAddedToRepository_and_then NotificationIsSent()`

  Siguiendo la estructura "given-when-then", dada una instancia de BookService, mockeamos un libro (`book`), definimos el comportamiento de su método `getTitle(): String` y definimos el comportamiento del mock `bookRepository` al añadir un nuevo libro, devolviendo en este caso el libro dado como argumento. Si, realizadas estas acciones, tratamos de incluir un libro en la librería; verificamos que se haya realizado una llamada al método `save(book)` de BookRepository con nuestro libro como argumento, así como que el sistema de notificaciones haya mostrado por pantalla el mensaje esperado tras añadir un nuevo libro (en la salida por pantalla deberá aparecer el título del libro, el cual hemos determinado nosotros al previamente definir el comportamiento de `getTitle(): String`).

- `givenBookService_whenABookIsDeleted_thenIsDeletedFromBookRepository_an d_thenNotificationIsSent()`

  En este caso, antes de eliminar un libro de la librería, debemos asegurarnos de que trabajamos con un libro existente en ella. Siguiendo la estructura "given-when-then", dada una instancia de BookService, primero mockeamos un libro (`book`) y definimos no solo el comportamiento de `getTitle(): String`, sino también le asignamos un valor a su atributo `id` que no entre en conflicto con los id's de los libros ya presentes en la librería. Una vez definido el comportamiento del mock `bookRepository` exactamente igual que en el anterior test, procedemos a guardar el libro asegurándonos de que lo que vayamos a borrar exista dentro. Una vez cumplidas estas condiciones, si llamamos al método `delete(book.getId())` del SUT, verificamos que se haya llamado una vez al método de borrado de la clase BookRepository, que el método `existsById(book.getId()): boolean` devuelva

false al cotejar si existe un libro con el id del libro borrado y, finalmente, que el sistema de notificaciones haya mostrado por pantalla el mensaje esperado tras eliminar un libro.

2.2. Tests de la API REST

Dentro del proyecto Maven, la clase de tests unitarios `BookRestAPITest.java` pertenece al paquete `es.urjc.code.daw.library.test.rest`. Mediante el uso de la librería REST Assured implementamos pruebas automáticas del sistema de API REST.

- `givenRestAPI_whenABookIsAdded_thenGETGivesTheBookBack()`

  Siguiendo la estructura "given-when-then", añadimos un nuevo libro al repositorio mediante una petición de tipo POST. Nos quedamos con el id asignado para el libro que recibimos en la respuesta de la petición, para así poder recuperarlo posteriormente. Después, realizamos una petición de tipo GET para recuperar el libro que hemos introducido con el id que teníamos. Entonces el comportamiento esperado es que la petición haya sido un éxito mediante la devolución del código 200 y que el libro que recibimos contiene en su descripción parte de la descripción del libro que queríamos introducir.

- `givenRestAPI_whenABookIsDeleted_thenGETDoesNotGiveTheBookBack()`

  Siguiendo la estructura "given-when-then", añadimos un nuevo libro al repositorio mediante una petición de tipo POST, quedándonos de igual manera que antes con el id del libro que introducimos para posteriormente utilizarlo. Después borramos el libro del repositorio mediante una petición de tipo DELETE utilizando el id que obtuvimos anteriormente. Entonces cuando intentamos recuperar el libro mediante una petición de tipo GET utilizando ese mismo id, el comportamiento esperado es que recibamos un código de error 404 porque el libro ya no existe en el repositorio.

2.3. Tests de la interfaz web

Dentro del proyecto Maven, la clase de tests unitarios `BookWebInterfaceTest.java` pertenece al paquete `es.urjc.code.daw.library.test.web`. Se ha optado por usar el navegador Google Chrome para la ejecución de los tests, por lo que hemos utilizado el *WebDriver* de Selenium para Chrome (*ChromeDriver*). Tras lanzar la aplicación *SpringBoot* en un puerto aleatorio, con la previa configuración del driver y creación del browser, se ejecutan cada uno de los tests requeridos en métodos independientes:

- `whenBookIsAdded_thenBookCanBeFound()`

  Siguiendo la estructura "given-when-then", una vez establecida la conexión con la interfaz web en el puerto aleatorio, valiéndonos de *XPath* buscamos el único botón presente en la página de inicio y lo clickeamos para posteriormente, en la página a la que accedemos, rellenar los *text boxes* con un título y una descripción y proceder a añadir un libro en el almacén, usando de nuevo *XPath* para encontrar el botón deseado. Una vez realizamos estos pasos, nos encontramos en la página del libro recién añadido, y para volver a la pantalla de inicio (botón *"All Books"*) guardamos en este caso en una lista todos los elementos cuya ruta nos indica que son botones y hacemos click en el que se encuentra en la tercera posición. Si todo ha ido correctamente, al realizar un `assertNotNull(driver.findElement(By.partialLinkText("Hansel")))` comprobamos que sí que existe en la lista de libros uno cuyo título contiene parte del *String* que utilizamos al añadir el libro.

- `whenBookIsDeleted_thenCannotBeFound()`

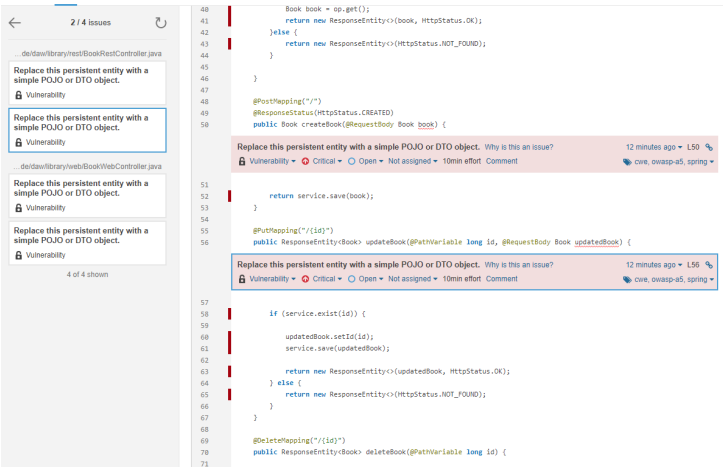
Siguiendo la estructura “given-when-then”, una vez establecida la conexión con la interfaz web en el puerto aleatorio, nos aseguramos en primer lugar de que lo que vayamos a eliminar ya esté en el repositorio realizando exactamente los mismos pasos que en el anterior test (son tests independientes, pero aún así hemos elegido otro nombre para evitar confusiones o conflictos). Una vez situados en la página del libro recién añadido, con el mismo método del test anterior guardamos en una lista todos los elementos de la página cuya ruta nos indica que son botones, para proceder a clickar el primer botón (*“Remove”*). Para comprobar que todo ha ido bien, guardamos en otra lista todos aquellos elementos cuyo link en la vista HTML nos indica que son libros del repositorio mediante la condición de búsqueda `By.partialLinkText("books")`. Finalmente, nos aseguramos de que en dicha lista no haya ningún libro cuyo título coincida con el título del libro eliminado.

3. Análisis estático de código

Antes de tratar los avisos reportados por SonarQube, mencionar que la existencia de la clase *LoginWebController* reportaba dos **“Security Hotspot”**. Al no usarse esta clase, y siguiendo las indicaciones del foro, se ha eliminado.

3.1. Issues de tipo “Vulnerability”

Cuando ejecutamos SonarQube nos encontramos con un total de 4 issues del tipo “Vulnerability”. Estos issues son todos del mismo tipo, **Persistent entities should not be used as arguments of "@RequestMapping" methods**, como podemos ver en la siguiente captura



Este issue se produce porque los objetos persistentes (en este caso los anotados como `@Entity`) no deberían ser pasados a métodos anotados con `@RequestMapping`, ya que son vulnerables a ataques maliciosos que pueden cambiar el contenido de campos inesperados de la base de datos.

Para resolver este *issue* hemos creado la clase *BookDTO*. Esta clase contiene los mismos atributos que la clase *Book*, como indica el patrón DTO. Esta nueva clase es aquella que se le pasa a los métodos del controlador en lugar de *Book*. Al no estar anotada como `@Entity`, una vez recibida, debemos hacer el mapping entre los atributos de las dos clases y guardar en el repositorio el objeto de la clase *Book*. De esta manera la vulnerabilidad quedaría resuelta.

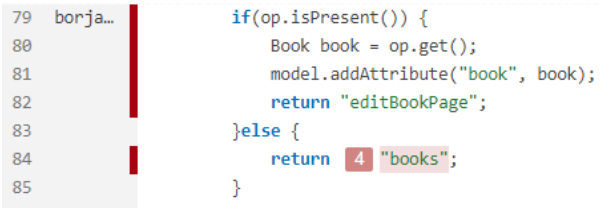
La estrategia seguida para la creación del *DTO* se ha basado en la documentación de las reglas de *SonarSource:* https://rules.sonarsource.com/java/tag/spring/RSPEC-4684.

3.2. Issues de tipo “Code Smell”

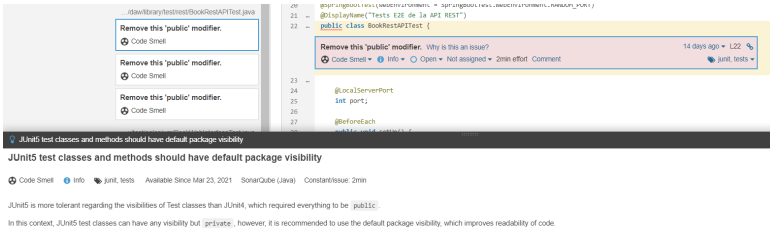
Una vez terminada la parte de los test, al ejecutar SonarQube nos encontramos con un total de 12 issues del tipo “Code Smell”. Entre ellos diferenciamos tres grandes grupos:

1. Un primer grupo, formado por un único *issue:* **Define a constant instead of duplicating this literal “books” 4 times.** La propia herramienta nos da feedback acerca del significado del problema y cómo solucionarlo, definiendo una constante nos ahorramos problemas en el futuro cuando haya que cambiar el literal. En el caso de

que hubiese que cambiar ese literal, con cómo está ahora habría que hacer cuatro modificaciones, mientras que con la declaración `static final String BOOKS = "books"` solo habría que realizar una.



2. Un segundo grupo, al cual pertenecen un total de 9 *issues* titulados **Remove this "public" modifier.** En las tres clases en las que se implementan tests (*BookServiceTest*, *BookWebInterfaceTest* y *BookRestAPITest*) la herramienta nos sugiere eliminar los modificadores de acceso justificando que en JUnit5 se recomienda la declaración de clases y métodos de test como "package". Debido a que este es el modificador de acceso por defecto en Java, para solucionar estos *Code Smell* basta con eliminar `public` de la cabecera de las mencionadas clases, así como de sus métodos test.



3. Un tercer y último grupo, con dos *issues* que en verdad hacen referencia a uno solo: **1 duplicated blocks of code must be removed.** Resulta que este "problema" en verdad es un **falso positivo**, ya que su origen radica en la aplicación del patrón de diseño DTO mencionado en el apartado 3.1 y no podemos evitar esta duplicación de código, siendo la aplicación de este patrón la técnica más adecuada. En base a lo expuesto, no se realizaron modificaciones en el código respecto a este reporte.